

# Π-Ware: Hardware Description and Verification in Agda\*

João Paulo Pizani Flor<sup>1</sup>, Wouter Swierstra<sup>2</sup>, and Yorick Sijsling<sup>3</sup>

- 1 Department of Information and Computing Sciences, Utrecht University,  
PO Box 80089, 3508 TB Utrecht, The Netherlands  
J.P.PizaniFlor@uu.nl
- 2 Department of Information and Computing Sciences, Utrecht University,  
PO Box 80089, 3508 TB Utrecht, The Netherlands  
W.S.Swierstra@uu.nl
- 3 Department of Information and Computing Sciences, Utrecht University,  
PO Box 80089, 3508 TB Utrecht, The Netherlands  
Y.Sijsling@uu.nl

---

## Abstract

There is a long tradition of modelling digital circuits using functional programming languages. This paper demonstrates that by employing *dependently typed programming languages*, it becomes possible to define circuit descriptions that may be simulated, tested, verified and synthesized using a single language. The resulting domain specific embedded language, Π-Ware, makes it possible to define and verify entire families of circuits at once. We demonstrate this by defining an algebra of parallel prefix circuits, proving their correctness and further algebraic properties.

**1998 ACM Subject Classification** B.7.2 Integrated Circuits: Design Aids – Verification, D.3.2 Programming Languages: Language Classifications – Functional Languages, F.3.1 Logics and Meanings of Programs: Specifying and Verifying and Reasoning about Programs

**Keywords and phrases** dependently typed programming, Agda, EDSL, hardware description languages, functional programming

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2015.9

## 1 Introduction

There is a long tradition of using functional programming to model hardware circuits. Dating as far back as the 1980's, there have been many different proposed Domain-Specific Languages (DSLs) for the design and verification of circuits [20, 2, 19, 5]. Initially these languages were mostly standalone, but later *embedded* DSLs for hardware were developed, hosted in functional languages such as Haskell or ML.

All of these *functional hardware* DSLs have some limitations with respects to (*type*) *safety* or aspects of the design process that they support. Most of them allow for simulation, some support synthesis to Register-Transfer Level (RTL) netlists; others are focused on verification (using a combination of SMT solvers, automated theorem provers, or interactive proof assistants). We would argue, however, that none of these are *unified* typed languages for the design, simulation, verification, and synthesis of hardware circuits.

---

\* This work was supported by the Netherlands Organization for Scientific Research (NWO) project on *A Dependently-Typed Language for Verified Hardware*.



Yet the need for better tools for the design of custom hardware accelerators is greater than ever. The performance of general purpose processors is becoming increasingly harder to improve, as we have already long ago faced the power wall and Instruction-Level Parallelism (ILP) shows diminishing returns [9]. There is a growing demand for hardware acceleration that is both easy to maintain and accurate.

This paper presents  $\Pi$ -Ware, a Hardware Description Language (HDL) embedded in *Agda* [18, 17], a dependently-typed programming language.  $\Pi$ -Ware provides a single, strongly-typed language which supports the definition, synthesis, simulation, testing and formal verification of complex circuits. The project is hosted at <http://piware.alvb.in>.

After giving a high-level overview of the language and its features (Section 2), this paper makes the following contributions:

- $\Pi$ -Ware has been designed to be a safe and strongly-typed language. Our embedding (Section 3) makes use of dependent types to provide safety guarantees beyond the ones offered by HDLs embedded in simply-typed functional languages. The embedding is parameterized by the type of fundamental data over the wires and the library of fundamental gates being used. For example, instead of using only logic gates, one could add binary arithmetic operators to the fundamental library. This raises the level of abstraction of the description and simplifies verification.
- Unlike other embeddings in proof assistants (such as Coquet [5]),  $\Pi$ -Ware circuits are executable. We defined functional semantics for both combinational and sequential circuits (Section 4). This may seem trivial, but providing a total semantics of circuits that run forever requires some care. Specifically, we define a *causal stream-based* semantics to model the simulation of sequential circuits.
- As the circuit semantics is executable, we can use it to test our designs. Furthermore, for finite domains, we can automatically derive a proof by exhaustive testing (Section 5). More generally, the full power of *Agda* as an interactive theorem prover can be used to prove properties of *circuit generators*. These generators are usually defined using some sort of recursive pattern (*connection pattern*), and proofs about them rely on induction following the same pattern (*proof combinator*).
- Finally, we show how all these features may be combined in a case study: the verification of parallel prefix circuits (Section 6). We describe this family of circuits in terms of  $\Pi$ -Ware primitives and verify its behaviour. In particular, we formulate and prove algebraic laws involving operators and transformations over parallel prefix circuits, providing machine-verified versions of proofs previously developed on paper [12].

## 2 Overview

We can illustrate circuit models in  $\Pi$ -Ware by analyzing a simple example. Let us model a 2-way multiplexer (**mux**). For convenience, booleans are being carried over the wires, and we have the usual set of gates at our disposal: NOT, AND, OR.

A first step when designing a circuit is to think of its *specification*. For such a small circuit as **mux**, its *truth table* (Table 1) serves well as specification.

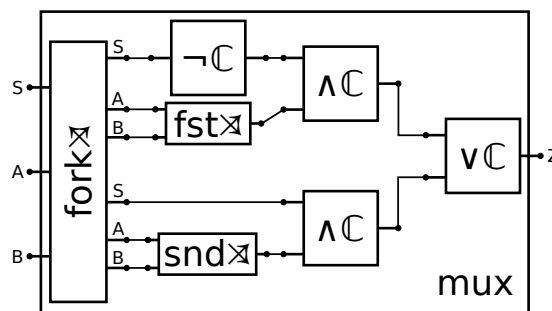
Our **mux** has two data inputs (**A**, **B**) and a selection input (**S**). It behaves in such a way that, whenever ( $S = 0$ ), the output is equal to input **A**, otherwise it is equal to input **B**. From the truth table we can (straightforwardly) derive a boolean formula:

$$Z = (A \wedge \neg S) \vee (B \wedge S) \tag{1}$$

From this logical formula, a designer could then *implement* a circuit with the structure shown in Figure 1. This kind of graphical model is often known as *block diagram*.

■ **Table 1** Truth table specification of `mux`.

S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



■ **Figure 1** Block diagram of `mux`.

We can also view such a diagram in a different way, by considering the fundamental gates present, and grouping them using *sequential* (`(_)>_`) and *parallel* (`(_)||_`) composition. This corresponds exactly to the definition of `mux` in  $\Pi$ -Ware, shown in Listing 1.

In this description, we use three kinds of fundamental gates: AND ( $\wedge C$ ), OR ( $\vee C$ ) and NOT ( $\neg C$ ). Notice how the circuit type is *indexed* by sizes of the input and output. The *total* size of all inputs of `mux` amounts to 3 and total size of all outputs amounts to 1. Besides fundamental gates and composition, we also use some blocks to do *rewiring*. The circuit called `fork $\times$  $\times$`  outputs two exact copies of its input bus side-by-side, while `fst $\times$  $\times$` <sub>1</sub> and `snd $\times$  $\times$` <sub>1</sub> select respectively the first and second wire from an input of size 2.

The `mux` example shows how  $\Pi$ -Ware circuits are described in a low level of abstraction. Circuits are combined in an *architectural* way, and there is no way *in the DSL* to refer to intermediary results (no variable binding). We discuss how this low-level description relates to other levels of abstraction in Section 7.

In `mux`, the parameters to the `C` type constructor are constants, but they need not be in general. Using dependent types, we can precisely define and reason about *circuit generators*. For example, here is the type and definition of the aforementioned `fork $\times$  $\times$` :

$$\begin{aligned} \text{fork}\times\times &: \forall \{n\} \rightarrow C\ n\ (n + n) \\ \text{fork}\times\times\ \{n\} &= \text{Plug}\ (\text{tabulate}\ \{n\}\ \text{id}\ ++\ \text{tabulate}\ \{n\}\ \text{id}) \end{aligned}$$

Even though circuit semantics is only given in Section 4, we can already take a look at what would be possible given a functional (simulation) semantics for our examples. For now, let's assume the semantic function for circuits has the following type:

$$\llbracket \_ \rrbracket : C\ i\ o \rightarrow (\text{Vec}\ \text{Bool}\ i \rightarrow \text{Vec}\ \text{Bool}\ o)$$

```

mux : C 3 1
mux = fork××
  >> (¬C || fst×₁ >> ∧C) || (id×₁ || snd×₁ >> ∧C)
  >> vC

```

■ **Listing 1** II-Ware model of `mux`.

That is, it takes a circuit with inputs of *size*  $i$  and outputs of *size*  $o$ , and returns its semantics: a function between appropriately-sized binary words. A first possibility to “reason” about circuit behaviour is to just *test* a circuit with some inputs and observe the produced outputs to gain confidence in its correctness. In the following snippet we give some test cases for `mux`. As using dependent types implies evaluation during type checking, we can formulate our tests as a type checking problem, requiring that our circuit will compute the required type by definition:

```

test1 : [ mux ] (false :: (true :: false :: [])) ≡ (true :: [])
test2 : [ mux ] (true :: (true :: false :: [])) ≡ (false :: [])

```

This approach works fine to check the correctness of the simple `mux`: just write one test for each line of the truth table. However, one cannot simply *test* circuit generators (such as `fork××`), as that would entail simulating a possibly infinite number of circuits. To definitely convince ourselves of the correctness of `fork××` we will want to *prove* a more general statement, such as:

$$\text{fork}\times\times\sqsubseteq++ : \forall n (w : \text{Vec Bool } n) \rightarrow [ \text{fork}\times\times \{n\} ] w \equiv w ++ w$$

Here we can regard the concatenation function (`_++_`) as a formal *specification* of `fork××`, and `fork×× $\sqsubseteq$ ++` as a *proof* that `fork××` complies with its specification. What the statement of `fork×× $\sqsubseteq$ ++` intuitively means is that the circuit has the *effect* of duplicating its inputs into its outputs. The proof of this statement is written by induction on  $w$ , and relies on auxiliary lemmas involving vector functions such as `_++_` and `tabulate`.

We discuss proofs of circuit (generator) properties more thoroughly in Section 5. In that section we also talk about a notion of *equivalence* between circuits and several algebraic properties of circuit constructors and *combinators*. As a prerequisite for verification, however, we first must define the syntax and semantics of circuits, in Sections 3 and 4.

### 3 Circuit Structure

The *syntax* of II-Ware models gives a low-level description of a circuit’s *architecture*, indicating how fundamental *gates* are connected to each other. This style approximates *block diagrams* drawn by hardware designers, but with a key distinction: in II-Ware, components are connected in a *nameless* fashion, without explicitly mentioning ports or wires.

As II-Ware is a *deeply-embedded* DSL, the syntax of the language is defined by a datatype (called `C`). Our DSL distinguishes between *combinational* and *sequential* circuits. In summary, sequential circuits can have *internal state*, while combinational ones do not. We denote this distinction by *indexing* the `C` type with an element of `IsComb`:

```

data IsComb : Set where σ ω : IsComb

```

```

infixl 4 _>>_; infixr 5 _||_
data C : {s : IsComb} → ℕ → ℕ → Set where
  Gate : ∀ (g : Gate) {s} → C {s} (#in g) (#out g)
  Plug : ∀ {i o s} → i × o → C {s} i o

  _>>_ : ∀ {i m o s} → C {s} i m → C {s} m o → C {s} i o
  _||_ : ∀ {i1 o1 i2 o2 s} → C {s} i1 o1 → C {s} i2 o2 → C {s} (i1 + i2) (o1 + o2)

  DelayLoop : ∀ {i o l} → C {σ} (i + l) (o + l) → C {ω} i o

```

■ **Listing 2** The circuit datatype ( $\mathbb{C}$ ).

We consider circuits indexed with the  $\omega$  value to be sequential, and those with  $\sigma$  to be combinational. The choice of name for these (one-letter) constructors is only partially arbitrary: we were motivated by the usage of  $\Sigma^\omega$  in mathematics to represent the set of all infinite sequences over a given alphabet  $\mathcal{S}$ . This distinction between combinational and sequential circuits results in some convenience: with the knowledge that a circuit has no state, its simulation semantics can be simpler – just a function between appropriately-sized vectors. And with a simpler semantics, we can also get simpler proofs.

Listing 2 shows the circuit type ( $\mathbb{C}$ ). The handling of the `IsComb` tag in each of the constructors tells us which sort of semantics (stateful or not) we need to have from the subparts in order to get the semantics of the whole circuit.

The constructors for sequential (`_>>_`) and parallel (`_||_`) composition, for example, *preserve* the `s` tag. This means that, to evaluate a circuit  $(c_1 \gg c_2)$  in a stateless way, *both*  $c_1$  and  $c_2$  need to be stateless (combinational). Equivalently, if any of the parts is stateful, only a stateful evaluation of the whole is allowed.

Besides `IsComb`, the circuit datatype ( $\mathbb{C}$ ) is also indexed by two natural numbers. These correspond, respectively, to the *total* number of input wires into the circuit and total number of output wires from the circuit. We were strongly influenced in our circuit syntax design choices by Coquet [5], especially in the usage of dependent types in the (`_>>_`) and (`_||_`) constructors to enforce sizing constraints.

In order to facilitate discussion of the constructors of  $\mathbb{C}$ , we categorize them as either *primitive* or *composite*: composite constructors take arguments of type  $\mathbb{C}$ , while primitive ones do not. First, we look at the primitive constructors:

- Circuits constructed with `Gate` are the smallest possible *with computational content*. The whole `PiWare.Circuit` module is parameterized by a *gate library* (detailed in Section 3.2), and by calling `Gate` we simply pick one of those gates to use as building block.
- The other primitive constructor is `Plug`, which is necessary due to the *nameless* fashion in which we compose circuits. Since it is impossible to refer to any specific circuit port we cannot, for example, map the “first” output of a circuit to the “second” input of another. Plugs are required to do *rewiring*, but they perform *no computation*.

The argument to the `Plug` constructor has type  $i \times o$ , and this is a synonym for a mapping from output wires (indices) to input wires (indices).

$$\begin{aligned} \_ \times \_ &: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Set} \\ i \times o &= \text{Vec} (\text{Fin } i) o \end{aligned}$$

Using such a mapping, no `Plug` can ever be built containing any information other than the origin of each output wire. An intuitive definition for  $(i \times o)$  would be  $(\text{Fin } o \rightarrow \text{Fin } i)$ , but we opted for the (first-order) `Vec` representation to get easier combination of plugs and easier proofs. Also, the first-order representation will make synthesis more straightforward.

The composite constructors in `II-Ware` represent ways in which smaller circuits can be connected to form a larger one. First, let us focus on the most interesting of them: `DelayLoop`. Both other composite constructors  $(\_)\_$  and  $(\_)\_$  preserve the `IsComb` index. The `DelayLoop` constructor, however, is an exception: it is the only way to build a sequential circuit given a combinational one as argument.

This single possible way to *introduce state* makes the definition of circuit semantics simpler and, as the name hints, we make sure to always introduce a *clocked delay* at each occurrence of `DelayLoop`. Thus we avoid *combinatorial loops* in the circuit, which can make circuit analysis much more complex [10]. The remaining composite constructors of `C` are:

- Sequential composition  $(\_)\_$ , which connects the output of one circuit to the input of another. The indices ensure that the sizes match up correctly.
- Parallel composition  $(\_)\_$ , that creates a combined circuit which has as inputs (resp. outputs) the inputs (resp. outputs) of *both* constituent subcircuits.

Careful indexing of sequential and parallel composition, together with the type of  $(\_)\_$ , ensure that some design mistakes are *prevented by construction*. Floating wires are forbidden by  $(\_)\_$ : in a term “ $c_1 \gg c_2$ ”, the output size of  $c_1$  needs to equal the input size of  $c_2$ . Also, because `Plug` takes a *function* from outputs to inputs, only one source can be assigned to each load (no short-circuits). Lastly, the *totality* of the argument to `Plug` ensures that no plug output can be left unassigned.

As already mentioned, our circuit syntax is strongly influenced by Coquet [5]. Some differences are the partitioning of circuits by the `IsComb` tag into *combinational* or *sequential*, the first-order `Plugs`, and the type of the `DelayLoop` constructor, which in our case does not allow nesting of state.

In `II-Ware`, circuits are parameterized both by the type of data travelling in the wires (an `Atomic` type) and by a set of fundamental `Gates` upon which all circuits are built. The first design choice taken when describing circuits in `II-Ware` is which `Atomic` type to use, so let’s start with that.

### 3.1 Atomic Types

Hardware descriptions in VHDL or Verilog, often model the information carried on the wires as bits. This stays close to a physical implementation and thus remains popular, however, sometimes it is useful to think of other types being carried in the wires. For example, an enumeration type better describes the possible states of a state machine. In `II-Ware`, all circuit descriptions are *parameterized* by the type of element carried over the wires.

Types that can be carried over ports and wires are called *atomic* types (Listing 3). Elements of such types are considered to have *no parts* and cannot be *inspected* by `II-Ware`.

Some simple examples of atomic types are: `Bool`, (named) enumerations and `Fin n` (for some  $n$ ). In order to be used as an atomic type, a given type must be *finite* and *inhabited*. These requirements are packed up in the `Finitelnhabited` record (Listing 4).

We witness the finiteness of `Atom` by a bijection with `Fin n`, and the `default` field shows that the type in question has at least one inhabitant. The reason to forbid empty types from being used as `Atom` lies in the semantics of `DelayLoop`.

```

record Atomic : Set1 where
  field Atom : Set
        enum : Finitelnhabited Atom

  open Finitelnhabited enum public
  W = Vec Atom

```

■ **Listing 3** The `Atomic` record.

```

record Finitelnhabited {l} (A : Set l) : Set l where
  field finite : Finite A
        default : A

  open Finite finite public

```

■ **Listing 4** The `Finitelnhabited` record.

We will cover circuit semantics with more detail in Section 4 but, in summary, each occurrence of `DelayLoop` prepends one extra element to the circuit’s output stream. This extra element will have type `Vec Atom n` (for arbitrary  $n$ ), and thus we need to have *at least one arbitrary value* of type `Atom` at our disposal (`default`). As a last remark, we make `W` a synonym for `Vec Atom n`. Thus in any context parameterized by an instance of the `Atomic` record, we can refer to words of atoms in a more convenient way.

A last detail to note is that the bijection between the `Atom` type and `Fin n` implies the existence of a decidable equality (and decidable setoid structure) for `Atom`. There is a function in the Agda standard library (called `eq?n`) that gives a decidable equality for any type  $A$ , provided there is an injection from  $A$  into `Fin n`. In our case, we pass to `eq?n` the injection obtained as a consequence of the bijection between `Atom` and `Fin n`. The relevant definitions are shown in Listing 5.

## 3.2 Fundamental Gates

The `mux` example from Listing 1 was built with the usual boolean gates (AND, OR, NOT). Instead of hardwiring this choice in the definition of `C`,  $\Pi$ -Ware allows users to choose their own collection of *fundamental gates*. These could be the boolean gates mentioned above, but also more complex circuits, such as muxes, registers, or arithmetic circuits, depending on the particular design.

To build a library of fundamental gates, users must define a suitable Agda record specifying the interface and semantics of each gate. This record (`Gates`) is shown in Listing 6.

First of all, the whole `Gates` module is parameterized by an instance of `Atomic`, thus fixing `W` and defining the type of elements that appear as inputs and outputs of gates.

The type of gate identifiers is stored in the `Gate` field, and there are functions that assign to each gate identifier a corresponding number of inputs (`#in`), number of outputs (`#out`), and specification function (`spec`). Notice the highly dependent type of `spec` and of `Gates` as a whole: the return type of `spec` depends on its  $g$  parameter and on the `#in` and `#out` fields. The type of `#in` and `#out` depends, in turn, on `Gate`.

The choice of fundamental gates strongly influences circuit correctness proofs: the correctness of each gate defined by the `Gates` record is *assumed* rather than proved.

```


$$\_ \stackrel{?}{=} \_ : \forall \{A : \text{Set}\} \{fA : \text{Finite } A\} \rightarrow \text{Decidable } \{A = A\} \_ \equiv \_$$


$$\_ \stackrel{?}{=} \_ \{fA\} = \text{let open Finite fA in eq?}^n \text{ injection}$$



$$\_ \stackrel{?}{=} A \_ : \text{Decidable } \{A = \text{Atom}\} \_ \equiv \_$$


$$\_ \stackrel{?}{=} A \_ = \_ \stackrel{?}{=} \_ \{ \text{FiniteInhabited.finite enum} \}$$


decSetoidA : DecSetoid _ _
decSetoidA = decSetoid  $\_ \stackrel{?}{=} A \_$ 

```

■ **Listing 5** Decidable equality for `Atom` derived via injection to `Fin n`.

```

record Gates : Set1 where
  field Gate      : Set
        #in #out  : Gate → ℕ
        spec      : ∀ g → (W (#in g) → W (#out g))

```

■ **Listing 6** The `Gates` record.

To perform boolean logic with our circuits, we will want to use any *functionally complete* set of boolean gates. A particularly simple such set is {NAND}, which contains only the negated AND gate. First, we must define how many input and output ports does each gate in the library have:

```

#in #out : NandGate → ℕ
#in   $\bar{\lambda}C' = 2$ 
#out   $\bar{\lambda}C' = 1$ 

```

Notice that the parameter of the `#in` and `#out` functions is of type `NandGate`. This is the type of *gate identifiers* in the library. We impose no requirements on a type to satisfy this role, but here `NandGate` is a simple enumeration type. Having defined the interface of each gate in our library (there is only one), we then define the specification function:

```

spec- $\bar{\lambda}C$  : W 2 → W 1
spec- $\bar{\lambda}C$  (x :: y :: []) = [ not (x ∧ y) ]

```

There are no restrictions imposed by `II-Ware` on which kind of gate should or should not be present in a library, and higher-level `Atomic` and `Gates` instances can make designs much simpler. For example, with an `Atomic` instance defined to represent 8-bit signed integers, there can be a useful `Gates` library containing some set of modular arithmetic operators over these integers.

As another example of gate library, `II-Ware` also includes `BoolTrio`, a gate library operating over booleans with three boolean operations (NOT, AND, OR) and two constant gates (FALSE and TRUE). We specify the behaviour of the gates using the boolean functions from Agda’s standard library (`Data.Bool`).

When *simulating* a `II-Ware` circuit, we will use the specification functions of the gate library used in that circuit. Likewise, in proofs of circuit correctness, the fundamental gates are assumed to be correct. Therefore the elements in a `Gates` library can be seen as fundamental in two ways:



- Fundamental *behaviour*, as they have no subparts.
- Fundamental *functional correctness*, as it is assumed.

### 3.3 Abstraction Levels

Throughout this paper, we will deal with circuit models and circuit semantics only in terms of their *size* (the `C` datatype is indexed by two natural numbers, representing the sizes of a circuit's input and output). However, `II-Ware` offers a thin *data abstraction* layer, allowing Agda types in circuit's inputs/outputs (instead of `Vec Atom`).

A typed circuit is defined as just a wrapper record around a sized circuit (`C`). Therefore, the computation still is performed over words, but the description of a typed circuit contains information on how to *convert* between elements of the involved Agda types and the correspondingly-sized words.

This thin layer makes mainly *simulation* and testing more convenient and less verbose (no need to always build vectors to compare with in testing, for example). However, as the computation is still performed over words, proving a circuit's correctness will still rely on lemmas involving the *sized* level (vectors, atoms).

We discuss with more detail in Section 7 how this layer of data abstraction influences modelling and verification, and what could be other possible ways of raising the level of abstraction in circuit description.

## 4 Circuit Semantics

Due to the choice of using deep embedding to implement our DSL, it is possible to write several different semantics for circuit models.

When talking about deeply embedded languages, a semantic function is just a function mapping the Abstract Syntax Tree (AST) of our DSL to a desired *carrier* type. All of the circuit semantics currently implemented in `II-Ware` are *compositional*, which means that they can be defined by *folding* the `C` type with an algebra.

The module `PiWare.Circuit.Algebra` defines the *algebra type* for `C` (as a `record`), along with the associated *catamorphism* (`fold`). There are two algebra types: one for combinational circuits (`CσA`) and one for (possibly) sequential ones (`CA`). The only difference between them is that a case for `DelayLoop` is absent from `CσA`. Here we show the algebra type for combinational circuits (`CσA`):

```
record CσA : Set where
  field GateA : ∀ g#                → T (#in g#) (#out g#)
       PlugA  : ∀ {i o} → i × o → T i o
       _>>A_  : ∀ {i m o} → T i m → T m o → T i o
       _||A_  : ∀ {i1 o1 i2 o2} → T i1 o1 → T i2 o2 → T (i1 + i2) (o1 + o2)
```

We use Agda's feature of *sections* (parameterized anonymous modules) to avoid repetition of parameters used in several definitions. Firstly, the algebra record type (`CσA`) is parameterized by the *carrier* type (called `T`). Also, the catamorphism for combinational circuits (called `cataCσ`) is parameterized by an instance of the algebra record.

Listing 7 shows the catamorphism for combinational circuits (`cataCσ`). Notice how this definition, *by itself*, takes only the circuit to be interpreted as parameter. However, `cataCσ` is part of a section, so outside of the section it takes an extra parameter (an element of the record type `CσA`). Notice also how (due to the `σ` index) there is no need to define a clause for `DelayLoop`.

```

cataCσ : ∀ {i o} → C {σ} i o → T i o
cataCσ (Gate g) = GateA g
cataCσ (Plug f) = PlugA f
cataCσ (c₁ » c₂) = cataCσ c₁ »A cataCσ c₂
cataCσ (c₁ || c₂) = cataCσ c₁ ||A cataCσ c₂

```

■ **Listing 7** Catamorphism for combinational circuits.

```

simulationσ : CσA
simulationσ = record { GateA = spec
; PlugA = λ p ins → tabulate (flip lookup ins ∘ flip lookup p)
; _»A_ = flip _o'_
; _||A_ = λ f₁ f₂ → uncurry' _+_+ ∘ map× f₁ f₂ ∘ splitAt' _ }

```

■ **Listing 8** Simulation semantics algebra for combinational circuits.

## 4.1 Combinational Simulation

As a particular example of such a compositional semantics, we defined *executable* simulation for  $\Pi$ -Ware circuit models, which maps circuits to the domain of Agda functions. This semantics is *executable* in the sense that, by applying the function obtained using the semantics to an input, the same output should be calculated as if the circuit had been implemented in hardware and run.

When getting the simulation semantics of a combinational circuit, we want to obtain a function between appropriately-sized words, that is, a circuit of type “ $C\ i\ o$ ” should result in a function of type “ $W\ i \rightarrow W\ o$ ”. Thus the carrier type for the combinational simulation algebra is:

```

W→W : ℕ → ℕ → Set
W→W m n = W m → W n

```

With the appropriate carrier defined, we get a very simple type and definition for the combinational simulation function:

```

[[_]] : ∀ {i o} → C {σ} i o → W→W i o
[[_]] = cataCσ simulationσ

```

Notice how the type of the semantic function *requires* the interpreted circuit to be combinational (it must be indexed by  $\sigma$ ). In this way, the algebra used (`simulationσ`) does not have a field for the `DelayLoop` case. We show on Listing 8 the definitions for each of the fields in the algebra (`simulationσ`)<sup>1</sup>.

The cases for sequential (`_»A_`) and parallel composition (`_||A_`) rely, respectively, on function composition and `map×` over products. Sequential circuit composition is evaluated simply as flipped composition of the functions obtained from evaluating the subcircuits. For parallel composition, the simulation behaviour is to split the input at the appropriate index, pass each of the parts to the functions obtained from evaluating the subcircuits, and

<sup>1</sup> It is useful to note Agda’s convention of adding primes to the names of non-dependent functions (`uncurry'`, `_o'_`)

concatenate the results. In the case of fundamental gates, we simply rely on that gate’s specification function. This leaves the most interesting definition to be explained: `PlugA`.

In the case of a `Plug`, we build the output word *pointwise* by using `tabulate`. The `tabulate` function from Agda’s standard library “fills” a  $(\text{Vec } A \ n)$  by evaluating a given function of type  $(\text{Fin } n \rightarrow A)$  on all points of its domain. In our case, each of these points is an output index (element of  $\text{Fin } o$ ). First, we `lookup` the output index in the plug mapping  $p$ , obtaining the corresponding input index. Then we use this index to `lookup` the input word and place the correct `Atom` on the output.

Let’s now consider a simple example circuit, its simulation semantics and the involved types, to better understand how all these definitions fit into place. We consider a two-input NAND gate ( $\bar{\wedge}C$ ), with the following type and definition:

$$\begin{aligned} \bar{\wedge}C &: \forall \{s\} \rightarrow C \{s\} \ 2 \ 1 \\ \bar{\wedge}C &= \wedge C \gg \neg C \end{aligned}$$

The gate is described using two-input conjunction ( $\wedge C$ ) and one-input negation ( $\neg C$ ) as building blocks, and these pieces come from the library of fundamental gates that we are using (`BoolTrio`). By evaluating  $\bar{\wedge}C$  we then obtain the following function:

$$\begin{aligned} [\bar{\wedge}C] &: W \rightarrow W \ 2 \ 1 \\ [\bar{\wedge}C] &= \text{spec } \neg C' \circ \text{spec } \wedge C' \end{aligned}$$

To simulate  $\bar{\wedge}C$  we rely on the specification functions of both gates and on function composition. The names  $\neg C'$  and  $\wedge C'$  are just the gate *identifiers*. If we reduce the expression above further (expanding `spec` and  $W \rightarrow W$ ), we obtain the following:

$$\begin{aligned} [\bar{\wedge}C] &: W \ 2 \rightarrow W \ 1 \\ [\bar{\wedge}C] &= \lambda \{ (x :: y :: []) \} \rightarrow [\text{not } (x \wedge y)] \end{aligned}$$

The verification of circuits and circuit generators will be discussed in detail in Section 5. But it is already clear that it will rely heavily on laws involving vectors, as well as algebraic properties of the fundamental gates and plugs used in the design.

## 4.2 Sequential Simulation

*Sequential* circuits are those in which their output at any given instant may depend not only on a combination of the input at the same instant, but on the *sequence* of previous inputs.

In  $\Pi$ -Ware, we model only the *discrete time domain*, and therefore a circuit’s input *signal*<sup>2</sup> is *piecewise constant* (as well as its output signal). Because of this characteristic, we can model both input and output signals as `Streams`<sup>3</sup>, in which each element of the `Stream` is a word.

Perhaps the simplest example of a circuit with internal state is `shift`. This circuit will output at any clock cycle  $t$  the value present on its input at the preceding cycle. The architecture of `shift` consists simply of one `DelayLoop` and one `Plug`:

$$\begin{aligned} \text{shift} &: C \{\omega\} \ 1 \ 1 \\ \text{shift} &= \text{DelayLoop } \text{swap} \times_1 \end{aligned}$$

<sup>2</sup> By signal we mean a function over the time domain.

<sup>3</sup> A stream is an infinite list, i.e., a list without the `Nil` constructor.

The expected behaviour of `shift` exemplifies why the type of `Atoms` (values that can be carried over II-Ware wires) needs to be not only finite but also *inhabited*. The `shift` circuit will put out the value present at its input on the *previous* clock cycle. On the zeroth clock cycle, however, there is no *previous* cycle, and thus there must be some `default` value to be put out.

To discuss the semantics of `shift`, we first note that the type of `shift` is tagged by  $\omega$  (omega), and for circuits with  $\omega$  in their type, we must use the *sequential simulation semantics* ( $\llbracket \_ \rrbracket \omega$ ). In the specific case of `shift`, the function obtained via the sequential simulation semantics will have the following type:

$$\llbracket \text{shift} \rrbracket \omega : \text{Stream } (W \ 1) \rightarrow \text{Stream } (W \ 1)$$

The function obtained via  $\llbracket \_ \rrbracket \omega$  consumes and produces a `Stream` of adequately-sized words. To explain in detail how the sequential semantics is actually defined, however, we have to mention a key distinction between digital circuits and stream functions in general: An unconstrained stream function (that is, an arbitrary element of type `Stream A → Stream B`) can (possibly) *look into the future*. Considering `Streams` over the discrete time domain, one simple example of stream function that “looks into the future” is `tail`.

$$\begin{aligned} \text{tail} &: \forall \{A\} \rightarrow \text{Stream } A \rightarrow \text{Stream } A \\ \text{tail } (in_0 :: in_{1+}) &= \flat in_{1+} \end{aligned}$$

The element at position `0` in the output of `tail` depends on the input at position `1`, and so forth. Sequential circuits clearly cannot show this behaviour (at least not if we want to physically implement them). As we want our sequential circuits to be synthesizable to actual hardware, we should ensure that our semantics will only ever produce *causal stream functions*.

One way to define a causal stream function is as the unfolding of a function producing only the *next* output, given the current and past inputs. We call these functions *causal step functions*, and they are defined as follows:

$$\begin{aligned} \_ \Rightarrow^c \_ &: \forall \{l_1 \ l_2\} (A : \text{Set } l_1) (B : \text{Set } l_2) \rightarrow \text{Set } (l_1 \sqcup l_2) \\ A \Rightarrow^c B &= \Gamma c \ A \rightarrow B \end{aligned}$$

The symbol  $\Gamma c$  means *causal context*, and it is defined simply as a non-empty list (`List+`), that is, a pair of the head (current value) with a possibly-empty tail (past values).

$$\begin{aligned} \Gamma c &: \forall \{l\} (A : \text{Set } l) \rightarrow \text{Set } l \\ \Gamma c &= \text{List}^+ \end{aligned}$$

Coming back to the definition of simulation semantics for sequential circuits, we can now establish the *carrier* type for the algebra of sequential circuits as being a *causal step function* between words of the appropriate length. Then, the *causal* simulation of a circuit is defined as a catamorphism over  $\mathbb{C}$  with the `simulationc` algebra:

$$\begin{aligned} W \Rightarrow^c W &: \forall i \ o \rightarrow \text{Set} \\ W \Rightarrow^c W \ i \ o &= W \ i \Rightarrow^c W \ o \\ \llbracket \_ \rrbracket^c &: \forall \{i \ o\} \rightarrow \mathbb{C} \ i \ o \rightarrow W \Rightarrow^c W \ i \ o \\ \llbracket \_ \rrbracket^c &= \text{cataC } \{a\sigma = \text{simulation}\sigma\} \ \text{simulationc} \end{aligned}$$

```

simulationc : CA {W→W} {W⇒cW}
simulationc = record { GateA      = λ g → GateAσ g ∘ head
                    ; PlugA      = λ f → PlugAσ f ∘ head
                    ; _>>A_     = λ f1 f2 → f2 ∘ map+ f1 ∘ pasts
                    ; _||A_     = λ f1 f2 → uncurry' _+_+ _ ∘ map× f1 f2 ∘ unzip+ ∘ splitAt+ _
                    ; DelayLoopA = λ { _ } { o } f → takev o ∘ delay o f }
where open CσA simulationσ using () renaming (GateA to GateAσ; PlugA to PlugAσ)

```

■ **Listing 9** Simulation semantics algebra for sequential circuits.

Listing 9 shows the `simulationc` record. Note how the definitions packed inside of `simulationσ` are made available for use by `open`-ing the corresponding module in the `where` block.

In the case of `GateA` and `PlugA` we simply take the *present* value from the causal context and pass it to the corresponding combinational field (`GateAσ` and `PlugAσ`), while the sequence (`_>>A_`) and parallel (`_||A_`) cases are a bit more involved.

To understand the `_>>A_` case, first notice that each of the parameters  $f_1$  and  $f_2$  is a causal step function. The `pasts` function takes the causal context and produces a (non-empty) list of all of its tails: essentially, each element from this list is a causal context *viewed from a given moment*. We then `map+` the  $f_1$  function over each of these pasts, producing a list in which each element is the *next output considering that given past*. Finally, the result of mapping is fed as the causal context of  $f_2$ .

The definition for the parallel case (`_||A_`) is somewhat similar to the combinational one. We also split the input at the appropriate point and use `map×` to apply  $f_1$  and  $f_2$  to each part of the product. However, `splitAt+` works pointwise over the causal context, thus the need to use `unzip+` in order to make the list of pairs into a pair of lists.

Perhaps the most important case in the sequential simulation algebra is `DelayLoopA`. The `delay` function transforms the regular word function (which takes  $l$  extra wires) into a causal step function, which depends on the history of inputs instead of the current state. According to this semantics, a circuit built with `DelayLoop` corresponds to a *Mealy machine*, where the state has size  $l$ , and the combinational circuit inside of it calculates *both* the next output and the next state.

By calling our causal semantic function (`[_ ]c`) over a circuit we obtain a causal *step function*. Then, by just unfolding this step function we obtain the *causal stream function* which is actually the user-facing type for the simulation of sequential circuits:

$$\begin{aligned}
[_ ]\omega &: \forall \{i\ o\} \rightarrow \mathbb{C} \ i \ o \rightarrow (\text{Stream } (W \ i) \rightarrow \text{Stream } (W \ o)) \\
[_ ]\omega &= \text{runc} \circ [_ ]c
\end{aligned}$$

In contrast with the type of `[_ ]` (the combinational semantics), the type of `[_ ]ω` makes no requirement on how the circuit parameter should be *indexed*. This means that the sequential semantics can be used to obtain a stream function from both sequential and combinational circuits. In the case of evaluating a combinational circuit using `[_ ]ω`, the obtained stream function just applies the calculation performed by the circuit *pointwise* on the stream (ignoring the past).

## 5 Verification

Π-Ware also allows for the *verification* of circuit models. The kind of properties that can be stated and verified depends on the semantics being used. With Π-Ware in its current form,

we can express mainly *functional* specifications, that is, those related to the input/output characteristics of the circuit. Furthermore, due to the embedding in a dependently-typed language, II-Ware allows for both *testing* of any specific circuit, as well as *proofs* of correctness for *circuit generators*.

Tests and proofs can be written which check constraints on the outputs or witness arbitrary relations between the inputs and outputs of a circuit. In particular, the Design Under Test (DUT) can be verified to have the same input/output behaviour as an Agda function *assumed to be correct*. Also, we have defined a notion of *extensional equivalence* between circuits, allowing us to prove algebraic properties of circuit constructors and combinators, as well as to define provably-correct semantics-preserving circuit transformations.

## 5.1 Testing

Testing can be used by a designer to *gain confidence* in the functional correctness of a model early in the design process, before attempting to write proofs in their full generality. Writing test cases can also be a useful way to capture requirements from whoever commissioned the circuit, thus aiding in *validation*.

Using only the simulation functions (`[[_]]` and `[[_]c`), *manual* test cases can already be written: this method is usually called *unit testing*. These are some examples of unit tests for a two-input `mux` (the leftmost boolean in the input vector being the *selection* bit):

```
test-mux1 : [[ mux ]] (false :: (true :: false :: [])) ≡ [ true ]
test-mux1 = refl

test-mux2 : [[ mux ]] (true :: (true :: false :: [])) ≡ [ false ]
test-mux2 = refl
```

Unit testing is useful, but we can do better, thus the focus of this subsection is on II-Ware's facilities to help *test automation*. For circuits with inputs and outputs of small size, verification via *exhaustive checking* is feasible, and our ultimate goal is to make this as automatic and concise as possible.

The first step of abstraction from manually-written test cases is to have an Agda function serving as specification of the circuit behaviour. This means that, for any possible input to the circuit, evaluating the function with this input will produce an output assumed to be correct.

Continuing with our `mux` example, let's check its correctness by comparing it with a specification function. First of all, the simulation semantics of `mux` has the following type:

$$[[ \text{mux} ]] : W\ 3 \rightarrow W\ 1$$

Looking at this type, and thinking about the expected behaviour of `mux` (*selecting* one of two inputs), a reasonable candidate for specification is as follows:

```
ite : W 3 → W 1
ite (s :: a :: b :: []) = [ if s then b else a ]
```

The first required task for automatic exhaustive checking is to *generate* all possible values of the circuit's input type. For this, we need the input type to have an instance of the `Finite` record, which embodies a *bijection* between the type in question and `Fin n`. The definition of `Finite` is shown in Listing 10.

```

record Finite {l} (A : Set l) : Set l where
  field #A      : ℕ
        mapping : A ↔ Fin #A

open Inverse' mapping public

```

■ **Listing 10** The `Finite` record.

There are some instances of `Finite` defined in the  $\Pi$ -Ware library for primitive types (amongst which `Bool`), along with products, sums and vectors. Specifically, the input type of `[[ mux ]]` is `W 3` (equal to `Vec Bool 3`), so the necessary `Finite` instance relies on the pre-defined instances for vectors and booleans.

Having a way to generate all values of a type, we can create a vector containing all of them. More interestingly, we can create a (heterogeneous) vector containing all of the *proofs* that each value satisfies a certain predicate: this vector will contain proofs  $\{P(x_1), P(x_2), \dots, P(x_n)\}$  for all the elements  $x_n$  of a type  $A$  given a certain predicate ( $P : A \rightarrow \text{Set}$ ).

By using its bijection with `Fin n`, we can have a  $\forall$ -introduction rule for any `Finite` type.

$$\forall\text{-Finite} : \forall \{l\} \{A : \text{Set } l\} \{P : A \rightarrow \text{Set}\} \{fin : \text{Finite } A\} \\ \{ps : \text{vec}\uparrow (\text{tabulate } (P \circ \text{from } \{fin\}))\} \rightarrow (\forall (x : A) \rightarrow P x)$$

The  `$\forall$ -Finite` function takes an *implicit* parameter ( $ps$ ) containing the aforementioned heterogeneous vector of proofs. This parameter can be implicit because each of the vector's elements reduces to `(tt : T)` (if the predicate holds), and the whole “vector” reduces to a nested pair of units. Agda's definition of both pairs and the unit type as a record, combined with the  $\eta$ -rule for records, ensure that the value of  $ps$  can be guessed.

Now, our final goal is to have exhaustive checking of circuit behaviour, so we need a  $\forall$ -introduction rule for the type of circuit inputs and outputs – `Vec Atom n`, abbreviated as `W n`. We know that `Vec A n` is `Finite` whenever  $A$  also is (proof omitted here), and combining this knowledge with the previous definition of  `$\forall$ -Finite` we arrive at the desired  `$\forall$ -W` rule ( $\forall$ -introduction for *words*):

$$\forall\text{-W} : \forall \{n\} \{P : \text{W } n \rightarrow \text{Set}\} \{ps : \text{vec}\uparrow (\text{tabulate } (P \circ \text{fromFinite } \{ \text{Finite-W} \}))\} \\ \rightarrow (\forall (w : \text{W } n) \rightarrow P w)$$

In particular, we can then simply use  `$\forall$ -W` to prove properties involving all possible inputs of a circuit. The property in which we are interested is whether a circuit and a given specification function *agree* on a certain input.

$$\_ \sqsubseteq? \_ \text{at} \_ : \forall \{i o\} (c : \mathbb{C} \{ \sigma \} i o) (f : \text{W } i \rightarrow \text{W } o) \rightarrow (\text{W } i \rightarrow \text{Set}) \\ c \sqsubseteq? f \text{at } w = \mathbb{T} \lfloor (\lfloor c \rfloor w) \stackrel{?}{=} \text{W } (f w) \rfloor$$

The statement  $(c \sqsubseteq? f \text{at } w)$  can be read as “ $c$  complies with  $f$  at input  $w$ ”. This relation relies on a decidable equality over output words of the checked circuit ( `$\_ \stackrel{?}{=} \text{W} \_$` ), and uses it to compare the results obtained by running the circuit simulation and the specification function. In the definition of  `$\_ \sqsubseteq? \_ \text{at} \_$` , the call to `[[ ]]` serves only to transform the value returned by  `$\_ \stackrel{?}{=} \text{W} \_$`  into a `Bool`, which is then further transformed by `T` into a `Set` (either `T` or `⊥`).

To put the last pieces of the puzzle together we call  $\mathbb{V}\text{-}\mathbb{W}$ , passing the relation just defined (partially applied) as the predicate to be exhaustively checked. In this way we obtain the function we ultimately wanted: `check`.

```
check :  $\forall \{i o\} (c : \mathbb{C} i o) (f : \mathbb{W} i \rightarrow \mathbb{W} o) \{ps : \text{vec} \uparrow (\text{tabulate } (c \text{ } \_? \text{ } f \text{ at\_} \circ \text{ fromW } i))\} \rightarrow c \text{ } \_? \text{ } f$ 
check c f {ps} =  $\mathbb{V}\text{-}\mathbb{W} \{P = c \text{ } \_? \text{ } f \text{ at\_}\} \{ps\}$ 
```

This function performs automatic exhaustive checking to verify that a circuit complies with a given specification. It is feasible to use `check` for verification of small circuits such as mux, or for small parts of bigger designs (parts with few ports). For bigger designs, and in particular to verify circuit *generators*, we need to resort to manually-written proofs.

## 5.2 Proofs

The key advantage brought to verification by using a dependently-typed language is that properties can be proven not only of any *specific* circuit, but of *circuit generators*. Circuit generators are *parameterized* definitions from which for each value of the parameter, a different circuit can be derived.

The term “circuit generator” itself comes from the Lava [2] EDSL, but the idea of parameterized definitions is *at least* as old as VHDL’s *generics* [13]. The parameters of these circuit generators are usually structural properties of the circuit, such as the *sizes* or amount of inputs and outputs of a circuit. Another example would be configuring how many clock cycles does the input get delayed in a shift register.

Usually, these definitions will be *recursive*, and thus the proofs of statements involving these generators will then be performed by induction. A circuit generator `muxN`, that selects between *two* inputs of size  $n$  each, has the following type and definition:

```
muxN :  $\forall n \{s\} \rightarrow \mathbb{C} \{s\} (1 + (n + n)) n$ 
muxN zero = nil
muxN (suc n) = adapt n  $\gg$  (mux || muxN n)
```

For a given value of the parameter  $n$ , this definition produces a circuit with input size  $1 + (n + n)$  (1 selection bit, plus  $n$  bits for each input) and output size  $n$ . The base case is a circuit with one input and `zero` outputs, and that matches the size of the empty plug (`nil`). In the recursive case, we connect the 2-input `mux` and the recursive call (`muxN n`) in parallel, and we need a `Plug` (called `adapt`) to make the right wires meet the right ports.

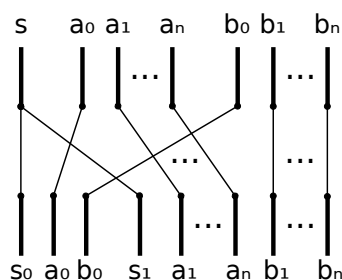
```
adapt :  $\forall n \{s\} \rightarrow \mathbb{C} \{s\} (1 + ((1 + n) + (1 + n))) ((1 + 1 + 1) + (1 + (n + n)))$ 
```

The diagram on Figure 2 shows exactly how `adapt` forks the selection bit and rearranges the remaining wires appropriately.

As already mentioned before, the choice of specification function has a significant impact on the proof of correctness for a circuit. In the case of `muxN`, the specification is `iteN`:

```
iteN :  $\forall n \rightarrow \mathbb{W} (1 + (n + n)) \rightarrow \mathbb{W} n$ 
iteN zero _ = []
iteN (suc n) ( _ :: ab ) with splitAt (suc n) ab
iteN (suc n) ( s :: .(a ++ b) ) | a , b , refl = if s then b else a
```





■ **Figure 2** Architecture of the `adaptX` plug.

In `iteN`, the tail of the input is split into two equal parts and we use `if_then_else_` to choose (based on the selection bit), which of the two parts will be the output.

The functional correctness property that we are interested in is the *pointwise equality* of the specification function for a circuit and the function obtained via the simulation semantics, that is, both functions have to *agree on all inputs*. This relation is abbreviated with the name `_⊆_`, where a statement  $(c \subseteq f)$  can be read as “*c complies with f*”.

In the case of our `muxN` example, the proof of compliance will need to follow the same induction pattern used to define the specification (`iteN`) itself. Namely, we need to pattern match on  $n$  and do case analysis on the result of splitting the tail of the input.

```

muxN⊆iteN : ∀ n → muxN n ⊆ iteN n
muxN⊆iteN zero   (_ :: []) = refl
muxN⊆iteN (suc n) (_ :: ab)      with splitAt (suc n) ab
muxN⊆iteN (suc n) (s :: .(a ++ b)) | a , b , refl = muxN⊆iteN'

```

Unfortunately, the full proof of `muxN⊆iteN` is a bit too long to be completely analyzed here (we abbreviate at `muxN⊆iteN'`). The proof relies of course on the proof of correctness for `mux` (the basic circuit with type `C 3 1`). It also relies on properties of the `adaptX` plug, ensuring that it’s semantics essentially rearranges the input word in the appropriate way.

### 5.3 Connection Patterns

We are interested not only in proving properties of circuits in isolation, but also about the behaviour of so-called *connection patterns*<sup>4</sup>. Connection patterns are just functions taking circuits as inputs and producing circuits as outputs. Typically they use the constructors of `C` to *connect* their arguments in a certain fashion, thus the name.

A (very simple) example of connection pattern is `parsN`, which connects  $n$  copies of a given circuit in parallel. The type and definition of `parsN` are:

```

parsN : ∀ {k i o s} → C {s} i o → C {s} (k * i) (k * o)
parsN {k} {i} {o} c = subst₂ C (*-sum-replicate k i) (*-sum-replicate k o)
                          (pars (replicate₂ {n = k} c))

```

Notice how the input and output sizes of the combined circuit are *statically guaranteed* to be correct, as they are calculated from the input/output sizes of the circuit passed as parameter. This definition (`parsN`) is a special case of a more general pattern: instead of

<sup>4</sup> This name comes from Lava as well.

replicating the same circuit  $n$  times, we can connect a whole vector of (different) circuits in parallel. This is achieved by `pars`:

```

pars : ∀ {n s} {is os : Vec ℕ n} (cs : Vec2 (C {s}) is os) → C {s} (sum is) (sum os)
pars {is = []}      {}      []2      = nilX
pars {is = _ :: _} {_ :: _} (c ::!2 cs) = c || pars cs

```

As the parameter of `pars` we need a special kind of vector, ensuring that only elements of types built with a given type constructor (`C`) can be present in the vector. This special kind of vector is `Vec2`, what we call an *index-heterogeneous vector*<sup>5</sup>. It is heterogeneous in the sense that its elements have different types, but only the indices vary, and the type constructor is fixed for all elements.

Another case of basic connection pattern is `seqsN`, taking a circuit and connecting  $n$  copies of it in sequence:

```

seqsN : ∀ k {s io} → C {s} io io → C {s} io io
seqsN k = seqs ∘ replicate {n = k}

```

Notice how the input and output sizes of the argument circuit are the same ( $io$ ). This is because the `__>>__` constructor forces the output size of a circuit in this sequence to match the input size of the next.

Also `seqsN` is a special case of a general pattern: connecting a vector with  $n$  circuits in sequence. For this connection to be even *possible*, we need the input/output sizes of the circuits in the vector to be *pairwise compatible*. This means that for each circuit, its output size must be equal to the input size of the next. To this end, we adapt the work done on *type-aligned sequences* [22] to a dependently-typed setting.

We are currently working on establishing lemmas about the behaviour of these connection patterns in order to make proofs involving their usage simpler. For example, the simulation behaviour of `seqsN` can be shown to be that of the `iterate` function, that is:

$$\forall w \rightarrow \llbracket \text{seqsN } k \ c \rrbracket w \equiv (\text{iterate } k \llbracket c \rrbracket) w$$

Furthermore, we are also working on expressing connection patterns as folds over the underlying (indexed heterogeneous) vectors, as that would allow for more general and powerful laws.

## 5.4 Circuit Equivalence

Until now we have talked about relations between a circuit and a function — such as the *complies with* relation (i.e. “ $c \sqsubseteq f$ ”). However, it is also very important to have an *equivalence relation* between circuits themselves. Given a properly-defined such relation, we can then have at our disposal laws like “ $c \gg \text{id}_X \approx c$ ”, allowing for *provably safe* circuit optimizations.

We have defined such a notion of circuit equivalence *up-to-simulation* for *combinational* circuits, and a similar notion (and laws) for sequential circuits is left for future work. In this section we explain the several iterations we have gone through until achieving the current definition of circuit equivalence, correcting a small issue at each step. In the most naïve and first attempt, we just require identical inputs and compare the outputs of simulating both circuits using (propositional) equality.

<sup>5</sup> More specifically, `Vec2` only handles type constructors with *two* indices.

$$\begin{aligned} \_ \equiv c \_ &: \forall \{i\ o\} (c_1\ c_2 : \mathbb{C}\ i\ o) \rightarrow \text{Set} \\ c_1 \equiv c_2 &= \forall w \rightarrow \llbracket c_1 \rrbracket w \equiv \llbracket c_2 \rrbracket w \end{aligned}$$

This definition is very unsatisfactory though, because it can only be used to compare circuits with *definitionally equal* indices, i.e, we cannot compare  $(c_1 : \mathbb{C}\ 1\ n)$  with  $(c_2 : \mathbb{C}\ 1\ (n + 0))$ . The first improvement over this definition is to use *vector equality* to compare the outputs. The notion of *semi-heterogeneous* vector equality ( $\_ \approx \_$ ) is defined in Agda’s standard library and it considers two vectors equal whenever the elements are *pointwise* propositionally equal. The new definition of circuit equivalence looks as follows:

$$\begin{aligned} \_ \approx \_ &: \forall \{i\ o_1\ o_2\} \rightarrow \mathbb{C}\ i\ o_1 \rightarrow \mathbb{C}\ i\ o_2 \rightarrow \text{Set} \\ c_1 \approx c_2 &= \forall w \rightarrow \llbracket c_1 \rrbracket w \approx \llbracket c_2 \rrbracket w \end{aligned}$$

While now the problem of word size  $(n + 0$  vs.  $n)$  has been solved for the outputs, the issue remains for the input: we cannot yet compare  $(c_1 : \mathbb{C}\ n\ 1)$  with  $(c_2 : \mathbb{C}\ (n + 0)\ 1)$ . Ultimately, what we want for circuit equivalence is to ensure that, when given “vector equal” inputs, both circuits will generate “vector equal” outputs:

$$\begin{aligned} \_ \approx \_ &: \forall \{i_1\ o_1\ i_2\ o_2\} \rightarrow \mathbb{C}\ i_1\ o_1 \rightarrow \mathbb{C}\ i_2\ o_2 \rightarrow \text{Set} \\ \_ \approx \_ &\{i_1\} \{ \_ \} \{i_2\} \{ \_ \} c_1\ c_2 = \\ &\forall \{w_1 : \mathbb{W}\ i_1\} \{w_2 : \mathbb{W}\ i_2\} \\ &\rightarrow w_1 \approx w_2 \rightarrow \llbracket c_1 \rrbracket w_1 \approx \llbracket c_2 \rrbracket w_2 \end{aligned}$$

This is the definition that *almost* gets us there. It has a big problem, though: it’s unsound. Very easily we can construct a term of type  $(c_1 \approx c_2)$  simply by making sure the hypothesis is false. A simple example of a term that should be banned by  $\_ \approx \_$  but is allowed is the following:

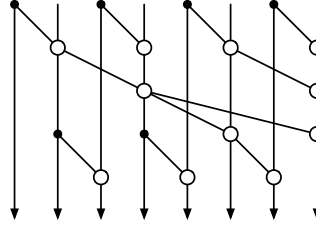
$$\begin{aligned} \approx\text{-unsound} &: (c_1 : \mathbb{C}\ 0\ 0) (c_2 : \mathbb{C}\ 1\ 1) \rightarrow c_1 \approx c_2 \\ \approx\text{-unsound} &c_1\ c_2\ () \end{aligned}$$

To solve this issue, we make an extra requirement for two circuits to be considered equal. Now, not only vector equal inputs must lead to vector equal outputs, but also there must be a proof that the sizes of the input words are propositionally equal.

$$\begin{aligned} \text{data } \_ \approx \_ &\{i_1\ o_1\ i_2\ o_2\} : \mathbb{C}\ i_1\ o_1 \rightarrow \mathbb{C}\ i_2\ o_2 \rightarrow \text{Set} \text{ where} \\ \text{refl} \approx &: \{c_1 : \mathbb{C}\ i_1\ o_1\} \{c_2 : \mathbb{C}\ i_2\ o_2\} (i \equiv : i_1 \equiv i_2) \\ &\rightarrow c_1 \approx c_2 \rightarrow c_1 \approx c_2 \end{aligned}$$

Besides the requirement over sizes of inputs, we also implement this relation as a datatype (instead of a `Set`-valued function). This allows us to pattern-match on the arguments of the `refl $\approx$`  constructor, which is needed when proving lemmas about  $\_ \approx \_$  (for example, symmetry and transitivity).

With  $\_ \approx \_$ , we arrive at the definition of circuit equivalence used to state algebraic properties of circuit constructors and combinators, and also in the case study discussed in Section 6. For extra convenience, we packed up  $(\mathbb{C}, \_ \approx \_)$  into an indexed setoid structure, and we added to Agda’s standard library some facilities for *equational reasoning* with indexed setoids. All in all, this allows proofs about circuit equivalence to be written in a very nice-looking style. A good example of such a proof can be seen in Listing 12 of Section 6.



■ **Figure 3** Example of an 8-input parallel prefix circuit.

## 6 Case Study: Parallel Prefix Circuits

In order to put in practice the definitions of  $\Pi$ -Ware we decided to perform a case study involving *parallel prefix circuits*. Parallel prefix circuits are a wide family of circuit architectures that compute *scans*, that is, given a binary operator  $\oplus$  and a vector of inputs  $[x_0, x_1, x_2, \dots, x_n]$ , it calculates the output vector  $[x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), \dots, (x_0 \oplus \dots \oplus x_n)]$ .

When talking about parallel prefix circuits, we always assume that the binary operator  $\oplus$  is *associative*, thus allowing different parts of the output to be calculated *in parallel*. In Figure 3 we show an example of a circuit utilizing maximal parallelism to calculate a scan with 8 inputs.

In this style of diagram, the data flows from top to bottom, each black dot is a forking point for wires and each white circle is an occurrence of the binary operator. Our case study was heavily influenced by the paper “An Algebra of Scans” [12]. In this paper, the author defines a set of primitives and combinators from which any scan circuit can be built, then proves their algebraic properties.

Our work consisted of formalizing the same primitives and combinators using  $\Pi$ -Ware, and proving the same basic algebraic facts about these combinators. Also, we formalized what exactly means to be a scan circuit, and proved that applying *scan combinators* to scan circuits will result in a scan.

Several of the primitives and combinators defined in the original paper [12] match exactly those present in  $\Pi$ -Ware, amongst them sequential ( $\_ \gg \_$ ) and parallel composition ( $\_ \parallel \_$ ) along with the identity plug ( $\text{id} \times$ ). This coincidence makes the case study especially fruitful, as several of the basic algebraic properties assumed in the original paper could be proved in  $\Pi$ -Ware. For example, sequential combination ( $\_ \gg \_$ ) forms a monoid of circuits, with  $\text{id} \times$  as identity:

$$\begin{aligned} \gg\text{-left-identity} &: \forall \{i o\} (c : \mathbb{C} i o) \rightarrow \text{id} \times \gg c \approx c \\ \gg\text{-left-identity} & c = \approx \approx (\text{from-}\equiv \circ \text{cong} [\text{c}] \circ \text{id} \times \text{-id}) \end{aligned}$$

$$\begin{aligned} \gg\text{-assoc} &: \forall \{i m n o\} (c_1 : \mathbb{C} i m) (c_2 : \mathbb{C} m n) (c_3 : \mathbb{C} n o) \rightarrow (c_1 \gg c_2) \gg c_3 \approx c_1 \gg (c_2 \gg c_3) \\ \gg\text{-assoc} & c_1 c_2 c_3 = \approx \approx (\text{from-}\equiv \circ \lambda \_ \rightarrow \text{refl}) \end{aligned}$$

In fact, the need to state and prove these basic algebraic laws was what led us to develop the notion of circuit equivalence (Section 5.4) in the first place. We predict that such algebraic structures over circuits will be important when reasoning about circuit transformations and synthesis. For example, the identity laws for  $\text{id} \times$  allow us to *remove* such plugs from any circuit, while being *certain* that the functional behaviour will not change.

```

serial : ∀ n → C n n
serial zero      = idX 0
serial (suc zero) = idX 1
serial (suc (suc n)) = serial (suc n) □ idX 1
    
```

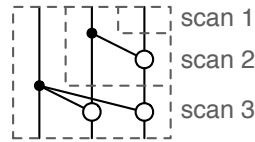
■ **Listing 11**  $\Pi$ -Ware description of a serial scan.

The concept of scan circuit itself was formalized by defining a “prototype” scan, which was assumed to be correct. This definition is very inefficient (in terms of gate usage and also in depth), but has a very simple inductive definition:

```

scan : ∀ n → C n n
scan zero = idX0
scan (suc n) = idX1 || scan n » fan (suc n)
    
```

Besides the parts already mentioned ( $\_ \gg \_$ ,  $\_ || \_$ ,  $\text{idX}$ ), here we also use the **fan** primitive. A term “**fan**  $n$ ” has type  $C\ n\ n$ , and calculates  $[x_0, (x_0 \oplus x_1), (x_0 \oplus x_2), \dots, (x_0 \oplus x_n)]$ . The diagram in Figure 4 illustrates the structure of “**scan** 3”.



■ **Figure 4** Structure of the prototype scan of size 3.

Using this specification, we could prove that several different architectures all indeed compute a scan. The proofs rely on the fact that all of these architectures are built by combining smaller scans into bigger ones.

Namely, we defined in  $\Pi$ -Ware the *sequential scan combinator* (called  $\_ \sqsupset \_$ ) and the *parallel scan combinator* (called  $\_ \sqcup \_$ ). The sequential scan combinator connects the last output of its first argument into the first input of the second argument. For the parallel combinator both scan circuits are put side-by-side, and an extra **fan** connects the last output of the first argument to all inputs of the second.

Having defined those combinators, we then proved that their definitions indeed satisfy their namesake property: whenever given scans as arguments, they produce a scan as output:

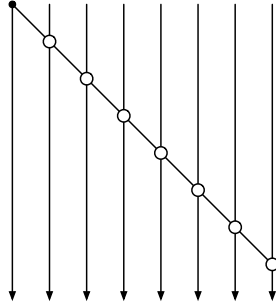
```

□-law : scan (suc m) □ scan (suc n) ≈ scan (m + suc n)
□-law : scan (suc m) □ scan n      ≈ scan (suc m + n)
    
```

As an example of a proof involving lots of these algebraic properties, we show the correctness of a *serial scan*. A serial scan is a parallel prefix circuit of maximal depth, as it makes no use of parallelism at all, and it has the structure shown in Figure 5.

The  $\Pi$ -Ware description for **serial** is pretty simple and makes essential use of the parallel scan combinator ( $\_ \sqcup \_$ ). The code for **serial** is shown in Listing 11

Finally, we prove that **serial** does indeed compute a scan. The proof (Listing 12) relies on the key fact that  $\_ \sqcup \_$  preserves scans. Furthermore, it relies on the fact that  $\_ \sqcup \_$  is a congruence with regards to circuit equivalence, and that two calls of **scan** with equal arguments will be equivalent (**scan-cong**).



■ **Figure 5** Structure of a serial scan.

```

serial-is-scan : ∀ n → serial n ≈ scan n
serial-is-scan zero           = ≈-refl
serial-is-scan (suc zero)    = id⊗1≈scan1
serial-is-scan (suc (suc n)) = begin
  serial (suc (suc n))
  ≈⟨ - definition of serial (suc (suc n))
  serial (suc n) ⊔ id⊗1
  ≈⟨ serial-is-scan (suc n) ⊔-cong id⊗1≈scan1 ⟩
  scan (suc n) ⊔ scan 1
  ≈⟨ ⊔-law n 1 ⟩
  scan (suc n + 1)
  ≈⟨ scan-cong (cong suc (+-comm n 1)) ⟩
  scan (suc (suc n))
  ■

```

■ **Listing 12** Proof that `serial` computes a scan.

## 7 Discussion

### 7.1 Related Work

There are numerous languages for hardware description; there is a wide variety of techniques that may be used for hardware verification, including the usage of automatic theorem provers, SAT solvers, model checking, and interactive theorem provers, notably HOL [7].

Systems such as ACL2 have been used to prove correctness of entire microprocessors [14], and the maturity of the ACL2 and HOL ecosystems is clearly visible in the highly optimized engines and large scale of some of the formalization efforts done using these languages. One of the key differences with our approach is the use of a typed higher-order host language, with which we can also have *higher-order specifications* for *connection patterns*. For example, the behaviour of the `parsN` pattern is equivalent to a functorial *map* over vectors.

The field of formal methods and functional programming applied to hardware design is indeed a crowded one, thus rather than attempt to survey these fields here, we will restrict ourselves to the most closely related work. There has been a great deal of work in the last thirty years marrying functional programming and hardware design, leading to languages such as Lava [2], Hawk [16], Wired [1] and ForSyDe [19]; Sheeran [21] gives an excellent

overview. When comparing  $\Pi$ -Ware to these other DSLs, the overarching theme is the use by  $\Pi$ -Ware of a host language in which stronger static guarantees are provided and in which circuits and proofs live side-by-side.

Specifically when comparing  $\Pi$ -Ware and Lava, we can say:

- Lava uses observational sharing as a binding technique, while  $\Pi$ -Ware is combinator-based.
- Lava verification uses external tools, while  $\Pi$ -ware circuits and proofs share a language.
- In Lava, only specific instances of a circuit generator can be verified, while  $\Pi$ -ware exploits the inductive structure of generators for verifying the generators themselves.

ForSyDe is still closely related to  $\Pi$ -Ware in terms of its goals, but much less closely than Lava. It offers similar verification facilities to those of Lava (using external model checking tools), so the same comparison applies. However, ForSyDe offers a different “front-end” to the user, by using reflection features of Haskell (namely quasi-quoters). Another considerable restriction on ForSyDe is that the set of types supported as inputs and outputs of circuits is very restricted (booleans, sequences, tuples), and cannot easily be extended, while  $\Pi$ -Ware is designed to be as general as possible regarding these choices.

With regards to  $\Pi$ -Ware and Wired, both languages describe the *architecture* of circuits, but while Wired definitions specify the exact geometry and placement of components,  $\Pi$ -Ware only talks about *topology*. We recognize the usefulness of geometric descriptions, but we have chosen to focus on topology as our initial goal is to prove properties involving (the preservation of) functional correctness and circuit transformations with a space vs. time trade-off.

When looking at the literature for hardware verification methods, we note that the idea of using dependent types for circuit description is not new and can be traced back as far as Hanna [11]. The paper “Constructing Correct Circuits” [4] gives a clear example of how dependent types can *tie together* specification and implementation. In this paper, the authors give a mapping between Peano naturals and binary numbers, then used to build a (ripple-carry) binary adder which is *correct by construction*. The approach taken in  $\Pi$ -Ware is significantly different. Rather than carry the functional specification of a circuit *in its type*, we clearly separate the construction, testing, and verification of circuits. This means that, for example, a designer can first simulate some instances of a design and get confidence in its correctness before trying to *prove* it. This greater degree of freedom may be particularly useful when exploring the design space, deferring the testing and verification effort until a satisfactory candidate design has been found.

Some of the most complete EDSLs for hardware (Coquet [5] and Fe-Si [6]) are hosted in the Coq theorem prover. Our design and implementation has been particularly inspired by Coquet. Both Coquet and  $\Pi$ -Ware use a similar *structural* and *nameless* description of circuits, parameterized by the type of gates. The most important difference between  $\Pi$ -Ware and Coquet, however, is that  $\Pi$ -Ware defines a *functional* semantics for circuits, while Coquet uses a *relational* semantics, i.e., the semantics are specified by defining a suitably indexed inductive data type. The choice of semantics style is crucial:  $\Pi$ -Ware circuits can be tested, simulated, and verified as any other Agda function.

Where Coq’s richer language for proof tactics may provide a great deal of automation, the functional semantics presented here reduces *for free*, without relying on the invocation of tactics or proof search. We expect to reap the benefits of a functional semantics while combining them with some proof-by-reflection techniques [23, 15]. Furthermore, we can use Agda features such as goal and context reflection, as well as solvers for algebraic structures (monoids, semirings, etc.) [3].

## 7.2 Future Work

### Equality Plugs

When explaining the behaviour of `Plugs` in Section 3, we said that they perform no computation. But further than that, some plugs in fact have also *no structural effect*. By this we mean plugs whose mapping is the identity function. They usually are an expression of arithmetic equalities over circuit indices, such as associativity:

$$\begin{aligned} \text{assoc}\times &: \forall \{a\} \{b\} \{c\} \rightarrow ((a + b) + c) \times (a + (b + c)) \\ \text{assoc}\times \{a\} \{b\} \{c\} &= \text{eq}\times (+\text{-assoc } a \ b \ c) \end{aligned}$$

The need to place such a `Plug` between two circuits is essentially an artifact of Intensional Type Theory (ITT). In the sequence constructor `(_)>>_`, the output index of the first parameter and input index of the second must be *definitionally* equal, that is, they must have the same normal form. If Agda had the *equality reflection rule*, then equalities involving indices could be used during type checking, and we would not need to insert *equality plugs*.

Right now we are investigating two approaches to make this issue less inconvenient. Firstly, we can use the *ring solver* from Agda’s standard library (coupled with reflection) to automatically solve index equalities and introduce the corresponding plugs whenever needed. Secondly, there was a recent addition to Agda of a language pragma called `REWRITE`, which allows for user-defined equalities to be added to Agda’s typing rules, essentially turning Agda into an Extensional Type Theory (ETT). We will investigate how the use of this pragma affects our library and examples.

### Functional Language

Even though we are using a functional language to model our circuits, the circuit description themselves are very low-level. In particular, we need to rewire intermediate results explicitly using our `Plug` constructors. While our style closely resembles the netlist representation of circuits, we would like to provide circuit designers with a more high-level, applicative interface.

One problem that we must address to do so, however, is that of observable sharing [8]. Any domain specific language for hardware description embedded in a general purpose functional language must, at some point, ensure that the sharing and recursion of the circuit definitions are not lost. Although various solutions do exist, these typically place a higher burden on the programmer through the necessity of explicit fixed-point and sharing combinators or rely on specific compiler support. We hope to find a satisfactory solution to this problem in the context of dependently typed programming languages such as Agda, and use this to define a more “functional” layer on top of the definitions presented here.

### Typed Circuits

While II-Ware rules out certain errors, such as short-circuits, we would like to investigate how to provide stronger static guarantees. So far, we have parameterized the type of circuits by the *size* of their inputs and outputs; we have started investigating how to parameterize circuits by their *type*.

For example, the type of a 2-input multiplexer would then become “`C^ (Bool × (Bool × Bool)) Bool`”, rather than the less informative “`C 3 1`”. To add extra type information to our circuits, we define a `record` wrapper for typed circuits (Listing 13).



```

record Ĉ {s : IsComb} (A B : Set) {i j : ℕ} : Set where
  constructor MkĈ
  field {α̂} : ↓W↑ A {i}
        {β̂} : ↓W↑ B {j}
  base : C {s} i j

```

■ **Listing 13** Typed Circuit type.

Here we require that the input and output types of our circuits are synthesizable – that is, they can indeed be represented in our simulation semantics (as vectors of atoms). By adding a series of *smart constructors* that produce and combine such typed circuits, we can provide a more convenient and type-safe interface to our library. We are currently extending our library with such type-safe definitions, including use of reflection to generate the required serializer/deserializer and proof that they are inverses.

## 8 Conclusion

With  $\Pi$ -Ware we have only started to explore the benefits that dependent types offer to digital circuit design.  $\Pi$ -Ware and the wider Agda ecosystem may not be mature enough yet to compete with some of the existing commercial tools and more mature prover technology; nonetheless we believe that the combination of the executable circuits, static types, and compositional proofs that  $\Pi$ -Ware offers form a novel point in the design space.

All the examples we have developed up to now, especially the case study on scan circuits, lead us to believe that this is indeed a fruitful avenue of study. By treating circuits as first-class objects in a dependently-typed language, we can reason about their behaviour and prove algebraic properties of relations, operators *over* circuits, and circuit generators. At the same time, we can simulate our designs and synthesize netlist descriptions. It should come as no surprise that type theory, a language of both computation and proof, provides a perfect setting for hardware verification and simulation.

**Acknowledgements** We would like to thank the helpful comments and suggestions of the attendants of the TYPES 2015 conference in Tallinn where we presented our initial results on  $\Pi$ -Ware. The participation in other venues such as for instance the Midlands Graduate School 2015 in Sheffield was also very fruitful in allowing discussions about the type-theoretical underpinnings of this work.

---

## References

- 1 Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In Dominique Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2005. doi:10.1007/11560548\_4.
- 2 Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.*, pages 174–184. ACM, 1998. doi:10.1145/289423.289440.

- 3 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - A functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009. doi:10.1007/978-3-642-03359-9\_6.
- 4 E. Brady, J. McKinna, and K. Hammond. Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types. In M. T. Morazán, editor, *Trends in Functional Programming 8*, pages 159–176. Intellect, 2007.
- 5 Thomas Braibant. Coquet: A Coq library for verifying hardware. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2011. doi:10.1007/978-3-642-25379-9\_24.
- 6 Thomas Braibant and Adam Chlipala. Formal verification of hardware synthesis. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2013. doi:10.1007/978-3-642-39799-8\_14.
- 7 A. Camilleri, M. Gordon, and T. F. Melham. Hardware verification using higher-order logic. Technical Report 91, University of Cambridge Computer Laboratory, 1986. URL: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-91.pdf>.
- 8 Koen Claessen and David Sands. Observable sharing for functional circuit description. In P. S. Thiagarajan and Roland H. C. Yap, editors, *Advances in Computing Science - ASIAN'99, 5th Asian Computing Science Conference, Phuket, Thailand, December 10-12, 1999, Proceedings*, volume 1742 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 1999. doi:10.1007/3-540-46674-6\_7.
- 9 Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In Ravi Iyer, Qing Yang, and Antonio González, editors, *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*, pages 365–376. ACM, 2011. doi:10.1145/2000064.2000108.
- 10 Morteza Fayyazi and Laurent Kirsch. Efficient simulation of oscillatory combinational loops. In Sachin S. Sapatnekar, editor, *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 777–780. ACM, 2010. doi:10.1145/1837274.1837470.
- 11 F. K. Hanna and N. Daeche. Dependent types and formal synthesis. *Philos. Trans. Phys. Sci. Eng.*, 339(1652):121–135, 1992. URL: <http://www.jstor.org/stable/54016>.
- 12 Ralf Hinze. An algebra of scans. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 186–210. Springer, 2004. doi:10.1007/978-3-540-27764-4\_11.
- 13 IEEE standard VHDL language reference manual: IEEE standard 1076-1987, 1988. doi:10.1109/ieeestd.1988.122645.
- 14 Warren A. Hunt Jr. *FM8501: A Verified Microprocessor*, volume 795 of *Lecture Notes in Computer Science*. Springer, 1994. doi:10.1007/3-540-57960-5.
- 15 Pepijn Kokke and Wouter Swierstra. Auto in Agda - programming proof search using reflection. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29*

- July 1, 2015. *Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 276–301. Springer, 2015. doi:10.1007/978-3-319-19797-5\_14.
- 16 John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within Haskell. In Didier Rémy and Peter Lee, editors, *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.*, pages 60–69. ACM, 1999. doi:10.1145/317636.317784.
  - 17 U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007. URL: <http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf>.
  - 18 Nicolas Oury and Wouter Swierstra. The power of pi. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 39–50. ACM, 2008. doi:10.1145/1411204.1411213.
  - 19 I. Sander and A Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Trans. Comput. Aided Des. Integ. Circ. Syst.*, 23(1):17–32, 2004. doi:10.1109/tcad.2003.819898.
  - 20 M. Sheeran. muFP, a language for VLSI design. In *Proc. 1984 ACM Symp. on LISP and Functional Programming, LFP '84*, pages 104–112. ACM, 1984. doi:10.1145/800055.802026.
  - 21 Mary Sheeran. Hardware design and functional programming: a perfect match. *J. UCS*, 11(7):1135–1158, 2005. doi:10.3217/jucs-011-07-1135.
  - 22 Atze van der Ploeg and Oleg Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 133–144. ACM, 2014. doi:10.1145/2633357.2633360.
  - 23 Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in Agda. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2012. doi:10.1007/978-3-642-41582-1\_10.